# 8 Randomness and secrets

***This chapter covers***

- What randomness is and why it's important
- Obtaining strong randomness and producing secrets
- The pitfalls of randomness

This is the last chapter of the first part of this book, and I have one last thing to tell you before we move on to the second part and learn about actual protocols used in the real world. It is something I've grossly neglected so far—randomness.

You must have noticed that in every cryptographic algorithm you've learned (with the exception of hash functions), you had to use randomness at some point: secret keys, nonces, IVs, prime numbers, challenges, and so on. As I was going through these different concepts, randomness always came from some magic black box. This is not atypical. In cryptography white papers, randomness is often represented by drawing an arrow with a dollar sign on top. But at some point, we need to ask ourselves the question, "Where does this randomness really come from?"

In this chapter, I will provide you with an answer as to what cryptography means when it mentions randomness. I will also give you pointers about the practical ways that exist to obtain randomness for real-world cryptographic applications.

**NOTE** For this chapter, you'll need to have read chapter 2 on hash functions and chapter 3 on message authentication codes.

## 8.1 What's randomness?

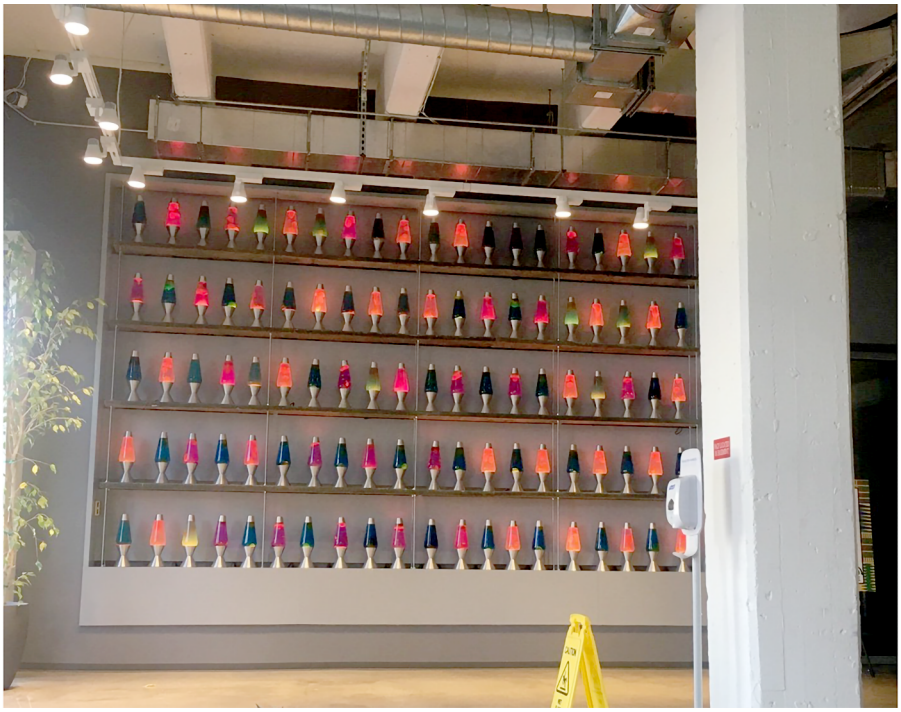
Everyone understands the concept of randomness to some degree. Whether playing with dice or buying some lottery tickets, we've all been exposed to it. My first encounter with randomness was at a very young age, when I realized that a RAND button on my calculator would produce a different number every time I pressed it. This troubled me to no end. I had little knowledge about electronics, but I thought I could understand some of its limitations. When I added 4 and 5 together, surely some circuits would do the math and give me the result. But a random button? Where were the random numbers coming from? I couldn't wrap my head around it.

It took me some time to ask the right questions and to understand that calculators actually cheated! They would hardcode large lists of random numbers and go through those one by one. These lists would exhibit good randomness, meaning that if you looked at the random numbers you were getting, there'd be as many 1s as 9s, as many 1s as 2s, and so on. These lists would simulate a *uniform distribution*: the numbers were distributed in equal proportions (uniformly).

When random numbers are needed for security and cryptography purposes, then randomness must be *unpredictable*. Of course, at that time, nobody would have used those calculators' "randomness" for anything related to security. Instead, cryptographic applications extract randomness from observing hard-to-predict physical phenomena.

For example, it is hard to predict the outcome of a dice roll, even though throwing a die is a deterministic process; if you knew all the initial conditions (how you're throwing the die, the die itself, the air friction, the grip of the table, and so on), you should be able to predict the result. That being said, all of these factors impact the end result so much that a slight imprecision in the knowledge of the initial conditions would mess with our predictions. The extreme sensitivity of an outcome to its initial conditions is known as *chaos theory*, and it is the reason why things like the weather are hard to predict accurately past a certain number of days.

The following image is a picture that I snapped during one of my visits to the headquarters of Cloudflare in San Francisco. LavaRand is a wall of lava lamps, which are lamps that produce hard-to-predict shapes of wax. A camera is set in front of the wall to extract and convert the images to random bytes.



Applications usually rely on the operating system to provide usable randomness, which in turn, gather randomness using different tricks, depending on the type of device it is run on. Common sources of randomness (also called *entropy sources*) can be the timing of hardware interrupts (for example, your mouse movements), software interrupts, hard disk seek time, and so on.

### Entropy

In information theory, the word *entropy* is used to judge how much randomness a string contains. The term was coined by Claude Shannon, who devised an entropy formula that would output larger and larger numbers as a string would exhibit more and more unpredictability (starting at 0 for completely predictable). The formula or the number itself is not that interesting for us, but in cryptography, you often hear "this string has low entropy" (meaning that it is predictable) or "this string has high entropy" (meaning that it is less predictable).

Observing interrupts and other events to produce randomness is not great; when a device boots, these events tend to be highly predictable, and they can also be maliciously influenced by external factors. Today, more and more devices have access to additional sensors and hardware aids that provide better sources of entropy. These

hardware random number generators are often called *true random number generators* (TRNGs) as they make use of external unpredictable physical phenomena like thermal noise to extract randomness.

The noise obtained via all these different types of input is usually not "clean" and sometimes does not provide enough entropy (if at all). For example, the first bit obtained from some entropy source could be 0 more often than not, or successive bits could be (more likely than chance) equal. Due to this, *randomness extractors* must clean and gather several sources of noise together before it can be used for cryptographic applications. This can be done, for example, by applying a hash function to the different sources and XORing the digests together.

Is this all there is to randomness? Unfortunately not. Extracting randomness from noise is a process that can be slow. For some applications that might need lots of random numbers quickly, it can become a bottleneck. The next section describes how OSs and real-world applications boost the generation of random numbers.

## 8.2 Slow randomness? Use a pseudorandom number generator (PRNG)

Randomness is used everywhere. At this point, you should be convinced that this is true at least for cryptography, but surprisingly, cryptography is not the only place making heavy use of random numbers. For example, simple Unix programs like ls require randomness too! As a bug in a program can be devastating if exploited, binaries attempt to defend against low-level attacks using a multitude of tricks; one of them is *ASLR* (address space layout randomization), which randomizes the memory layout of a process every time it runs and, thus, requires random numbers. Another example is the network protocol TCP, which makes use of random numbers every time it creates a connection to produce an unpredictable sequence of numbers and thwarts attacks attempting to hijack connections. While all of this is beyond the scope of this book, it is good to have an idea of how much randomness ends up being used for security reasons in the real world.

I hinted in the last section that, unfortunately, obtaining unpredictable randomness is somewhat of a slow process. This is sometimes due to a source of entropy being slow to produce noise. As a result, OSs often optimize their production of random numbers by using *pseudorandom number generators* (PRNGs).

> **NOTE** In order to contrast with random number generators that are not designed to be secure (and that are useful in different types of applications, like video games), PRNGs are sometimes called CSPRNGs for *cryptographically secure* PRNGs. NIST, wanting to do things differently (as usual), often calls their PRNGs *deterministic random bit generators* (DRBGs).

A PRNG needs an initial secret, usually called a *seed*, that we can obtain from mixing different entropy sources together and can then produce lots of random numbers quickly. I illustrate a PRNG in figure 8.1.

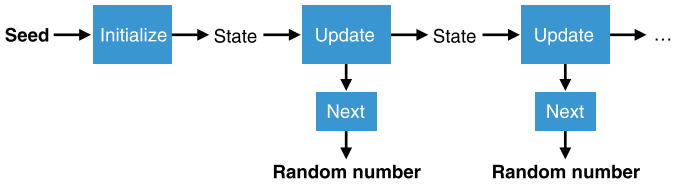



**Figure 8.1 A pseudorandom number generator (PRNG) generates a sequence of random numbers based on a seed. Using the same seed makes the PRNG produce the same sequence of random numbers. It should be impossible to recover the state using knowledge of the random outputs (the function `next` is one way). It follows that it should also be impossible from observing the produced random numbers alone to predict future random numbers or to recover previously generated random numbers.**

Cryptographically secure PRNGs usually tend to exhibit the following properties:

- *Deterministic*—Using the same seed twice produces the same sequence of random numbers. This is unlike the unpredictable randomness extraction I talked about previously: if you know a seed used by a PRNG, the PRNG should be completely predictable. This is why the construction is called *pseudo*random, and this is what allows a PRNG to be extremely fast.
- *Indistinguishable from random*—In practice, you should not be able to distinguish between a PRNG outputting a random number from a set of possible numbers and a little fairy impartially choosing a random number from the same set (assuming the fairy knows a magical way to pick a number such that every possible number can be picked with equal probability). Consequently, observing the random numbers generated alone shouldn't allow anyone to recover the internal state of the PRNG.

The last point is important! A PRNG simulates picking a number *uniformly at random*, meaning that each number from the set has an equal chance of being picked. For example, if your PRNG produces random numbers of 8 bytes, the set is all the possible strings of 8 bytes, and each 8-byte value should have equal probability of being the next value that can be obtained from your PRNG. This includes values that have already been produced by the PRNG at some point in the past.

In addition, many PRNGs exhibit additional security properties. A PRNG has *forward secrecy* if an attacker learning the state (by getting in your computer at some point in time, for example) doesn't allow the PRNG to retrieve previously generated random numbers. I illustrate this in figure 8.2.

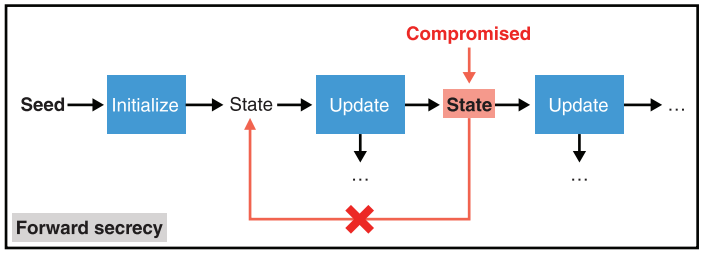



**Figure 8.2 A PRNG has forward secrecy if compromise of a state does not allow recovering previously generated random numbers.**

Obtaining the *state* of a PRNG means that you can determine all future pseudorandom numbers that it will generate. To prevent this, some PRNGs have mechanisms to "heal" themselves periodically (in case there was a compromise). This healing can be achieved by reinjecting (or re-seeding) new entropy after a PRNG was already seeded. This property is called *backward secrecy*. I illustrate this in figure 8.3.

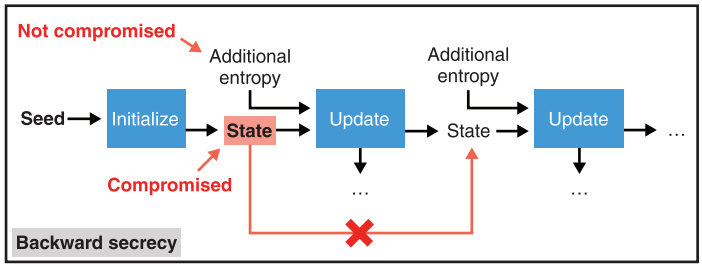



**Figure 8.3 A PRNG has backward secrecy if compromise of a state does not allow predicting future random numbers generated by the PRNG. This is true only once new entropy is produced and injected in the update function after the compromise.**

> **NOTE** The terms *forward* and *backward secrecy* are often sources of confusion. If you read this section thinking shouldn't forward secrecy be backward secrecy and backward secrecy be forward secrecy instead, then you are not crazy. For this reason, backward secrecy is sometimes called *future secrecy* or even *post-compromise security* (PCS).

PRNGs can be extremely fast and are considered safe methods to generate large numbers of random values for cryptographic purposes if properly seeded. Using a predictable number or a number that is too small is obviously not a secure way to seed a PRNG. This effectively means that we have secure cryptographic ways for quickly stretching a secret of appropriate size to billions of other secret keys. Pretty cool, right? This is why most (if not all) cryptographic applications do not use random

numbers directly extracted from noise, but instead use them to seed a PRNG in an initial step and then switch to generating random numbers from the PRNG when needed.

### The Dual-EC backdoor

Today, PRNGs are mostly heuristic-based constructions. This is because constructions based on hard mathematical problems (like the discrete logarithm) are too slow to be practical. One notorious example is *Dual EC*, invented by NSA, which relies on elliptic curves. The Dual EC PRNG was pushed to various standards including some NIST publications around 2006, and not too long after, several researchers independently discovered a potential backdoor in the algorithm. This was later confirmed by the Snowden revelations in 2013, and a year later the algorithm was withdrawn from multiple standards.

To be secure, a PRNG must be seeded with an *unpredictable* secret. More accurately, we say that the PRNG takes a key of $n$ bytes sampled uniformly at random. This means that we should pick the key randomly from the set of all possible $n$-byte strings, where each byte string has the same chance of being picked.

In this book, I talked about many cryptographic algorithms that produce outputs indistinguishable from random (from values that would be chosen uniformly at random). Intuitively, you should be thinking can we use these algorithms to generate random numbers then? And you would be right! Hash functions, XOFs, block ciphers, stream ciphers, and MACs can be used to produce random numbers. Hash functions and MACs are theoretically not defined as providing outputs that are indistinguishable from random, but in practice, they often are. Asymmetric algorithms like key exchange and signatures, on the other hand, are (almost all the time) not indistinguishable from random. For this reason, their output is often hashed before being used as random numbers.

Actually, because AES is hardware-supported on most machines, it is customary to see AES-CTR being used to produce random numbers. The symmetric key becomes the seed, and the ciphertexts become the random numbers (for the encryption of an infinite string of 0s, for example). In practice, there is a bit more complexity added to these constructions in order to provide forward and backward secrecy. Fortunately, you now understand enough to go to the next section, which provides an overview of obtaining randomness for real.

## 8.3 Obtaining randomness in practice

You've learned about the three ingredients that an OS needs to provide cryptographically secure random numbers to its programs:

- *Noise sources*—These are ways for the OS to obtain raw randomness from unpredictable physical phenomena like the temperature of the device or your mouse movements.

- *Cleaning and mixing*—Although raw randomness can be of poor quality (some bits might be biased), OSs clean up and mix a number of sources together in order to produce a good random number.
- *PRNGs*—Because the first two steps are slow, a single, uniformly random value can be used to seed a PRNG that can quickly produce random numbers.

In this section, I will explain how systems bundle the three concepts together to provide simplified interfaces to developers. These functions exposed by the OS usually allow you to generate a random number by issuing a system call. Behind these system calls is, indeed, a system bundling up noise sources, a mixing algorithm, and a PRNG (summarized in figure 8.4).

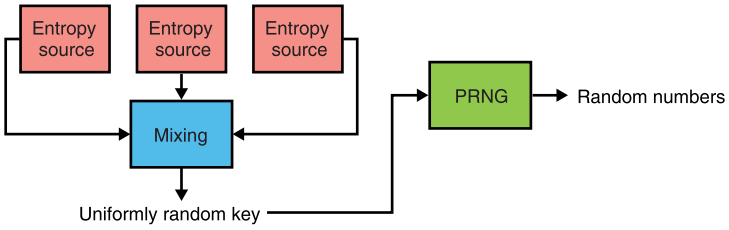



**Figure 8.4 Generating random numbers on a system usually means that entropy was mixed together from different noise sources and used to seed a long-term PRNG.**

Depending on the OS and on the hardware available, these three concepts might be implemented differently. In 2021, Linux uses a PRNG that's based on the ChaCha20 stream cipher, while macOS uses a PRNG that's based on the SHA-1 hash function. In addition, the random number generator interface exposed to developers will be different depending on the OS. On Windows, the `BCryptGenRandom` system call can be used to produce random numbers, while on other platforms, a special file (usually called /dev/urandom) is exposed and can be read to provide randomness. For example, on Linux or macOS, one can read 16 bytes from the terminal using the `dd` command-line tool:

```
$ dd if=/dev/urandom bs=16 count=1 2> /dev/null | xxd -p
40b1654b12320e2e0105f0b1d61e77b1
```

One problem with /dev/urandom is that it might not provide enough entropy (its numbers won't be random enough) if used too early after booting the device. OSs like Linux and FreeBSD offer a solution called `getrandom`, which is a system call that pretty much offers the same functionality as reading from /dev/urandom. In rare cases, where not enough entropy is available for initializing its PRNG, `getrandom` will block the continuation of the program and wait until properly seeded. For this reason,

I recommend that you use `getrandom` if it is available on your system. The following listing shows how one can securely use `getrandom` in C:

**Listing 8.1 Getting random numbers in C**

```
#include <sys/random.h>

uint8_t secret[16];
int len = getrandom(secret, sizeof(secret), 0);

if (len != sizeof(secret)) {
    abort();

}
```

Fills a buffer with random bytes (note that getrandom is limited to up to 256 bytes per call).

The default flags (0) is to not block, unless properly seeded.

It is possible that the function fails or returns less than the desired amount of random bytes. If this is the case, the system is corrupt and aborting might be the best thing to do.

With that example in mind, it is also good to point out that many programming languages have standard libraries and cryptographic libraries that provide better abstractions. It might be easy to forget that `getrandom` only returns up to 256 bytes per call, for example. For this reason, you should always attempt to generate random numbers through the standard library of the programming language you're using.

> **WARNING** Note that many programming languages expose functions and libraries that produce predictable random numbers. These are not suited for cryptographic use! Make sure that you use random libraries that generate *cryptographically strong* random numbers. Usually the name of the library helps (for example, you can probably guess which one you should use between the `math/rand` and `crypto/rand` packages in Golang), but nothing replaces reading the manual!

Listing 8.2 shows how to generate some random bytes using PHP 7. Any cryptographic algorithm can use these random bytes. For example, as a secret key to encrypt with an authenticated encryption algorithm. Every programming language does things differently, so make sure to consult your programming language's documentation in order to find the standard way to obtain random numbers for cryptographic purposes.

**Listing 8.2 Getting random numbers in PHP**

```
<?php
$bad_random_number = rand(0, 10);

$secret_key = random_bytes(16);
?>
```

Produces a random integer between 0 and 10. While fast, rand does not produce cryptographically secure random numbers so it is not suitable for cryptographic algorithms and protocols.

random_bytes creates and fills a buffer with 16 random bytes. The result is suitable for cryptographic algorithms and protocols.

Now that you've learned how you can obtain cryptographically secure randomness in your programs, let's think about the security considerations you need to keep in mind when you generate randomness.

## 8.4 Randomness generation and security considerations

It is good to remember at this point that any useful protocol based on cryptography requires good randomness and that a broken PRNG could lead to the entire cryptographic protocol or algorithm being insecure. It should be clear to you that a MAC is only as secure as the key used with it or that the slightest ounce of predictability usually destroys signature schemes like ECDSA, and so on.

So far, this chapter makes it sound like generating randomness should be a simple part of applied cryptography, but in practice, it is not. Randomness has actually been the source of many, many bugs in real-world cryptography due to a multitude of issues: using a noncryptographic PRNG, badly seeding a PRNG (for example, using the current time, which is predictable), and so on.

One example includes programs using *userland PRNGs* as opposed to *kernel PRNGs*, which are behind system calls. Userland PRNGs usually add unnecessary friction and if misused can, in the worst of cases, break the entire system. This was notably the case with the PRNG offered by the OpenSSL library that was patched into some OSs in 2006, inadvertently affecting all SSL and SSH generated keys using the vulnerable PRNG.

> *Removing this code has the side effect of crippling the seeding process for the OpenSSL PRNG. Instead of mixing in random data for the initial seed, the only random value that was used was the current process ID. On the Linux platform, the default maximum process ID is 32,768, resulting in a very small number of seed values being used for all PRNG operations.*
>
> —H. D. Moore ("Debian OpenSSL Predictable PRNG Toys," 2008)

For this reason and others, I will mention later in this chapter that it is wise to avoid userland PRNG and to stick to randomness provided by the OS when available. In most situations, sticking to what the programming language's standard library or what a good cryptography library provides should be enough.

> *We cannot keep on adding 'best practice' after 'best practice' to what developers need to keep in the back of their heads when writing everyday code.*
>
> —Martin Boßlet ("OpenSSL PRNG Is Not (Really) Fork-safe," 2013)

Unfortunately, no amount of advice can really prepare you for the many pitfalls of acquiring good randomness. Because randomness is at the center of every cryptography algorithm, making tiny mistakes can lead to devastating outcomes. It is good to keep in mind the following edge cases should you run into them:

- *Forking processes*—When using a userland PRNG (some applications with extremely high performance requirements might have no other choice), it is important to keep in mind that a program that forks will produce a new child process that will have the same PRNG state as its parent. Consequently, both PRNGs will

produce the same sequence of random numbers from there on. For this reason, if you really want to use a userland PRNG, you have to be careful to make forks use different seeds for their PRNGs.

- *Virtual machines (VMs)*—Cloning of PRNG state can also become a problem when using the OS PRNG. Think about VMs. If the entire state of a VM is saved and then started several times from this point on, every instance might produce the exact same sequence of random numbers. This is sometimes fixed by hypervisors and OSs, but it is good to look into what the hypervisor you're using is doing before running applications that request random numbers in VMs.
- *Early boot entropy*—While OSs should have no trouble gathering entropy in user-operated devices due to the noise produced by the user's interactions with the device, embedded devices and headless systems have more challenges to overcome in order to produce good entropy at boot time. History has shown that some devices tend to boot in a similar fashion and end up amassing the same initial noise from the system, leading to the same seed being used for their internal PRNGs and the same series of random numbers being generated.

> *There is a window of vulnerability—a boot-time entropy hole—during which Linux's urandom may be entirely predictable, at least for single-core systems. [. . .] When we disabled entropy sources that might be unavailable on a headless or embedded device, the Linux RNG produced the same predictable stream on every boot.*
>
> —Heninger et al. ("Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices," 2012)

In these rare cases, where you really, really need to obtain random numbers early during boot, one can help the system by providing some initial entropy generated from another machine's well-seeded `getrandom` or /dev/urandom. Different OSs might provide this feature, and you should consult their manuals (as usual) if you find yourself in this situation.

If available, a TRNG provides an easy solution to the problem. For example, modern Intel CPUs embed a special hardware chip that extracts randomness from thermal noise. This randomness is available through an instruction called `RDRAND`.

### The `RDRAND` controversy

Interestingly, Intel's `RDRAND` has been quite controversial due to the fear of backdoors. Most OSs that have integrated `RDRAND` as a source of entropy mix it with other sources of entropy in a way that is *contributory*. Contributory here means that one source of entropy cannot force the outcome of the randomness generation.

### Exercise

Imagine for a minute that mixing different sources of entropy was done by simply XORing them together. Can you see how this might fail to be contributory?

Finally, let me mention that one solution to avoid the randomness pitfalls is to use algorithms that rely *less* on randomness. For example, you saw in chapter 7 that ECDSA requires you to generate a random nonce every time you sign, whereas EdDSA does not. Another example you saw in chapter 4 is AES-GCM-SIV, which does not catastrophically break down if you happen to reuse the same nonce twice, as opposed to AES-GCM, which will leak the authentication key and will then lose integrity of the ciphertexts.

## 8.5 Public randomness

So far, I've talked mostly about *private randomness*, the kind you might need for your private keys. Sometimes, privacy is not required and *public randomness* is needed. In this section, I briefly survey some ways to obtain such public randomness. I distinguish two scenarios:

- *One-to-many*—You want to produce randomness for others.
- *Many-to-many*—A set of participants want to produce randomness together.

First, let's imagine that you want to generate a stream of randomness in a way that many participants can verify it. In other words, the stream should be unpredictable but impossible to alter from your perspective. Now imagine that you have a signature scheme that provides unique signatures based on a key pair and a message. With such a signature scheme, there exists a construction called a *verifiable random function* (VRF) to obtain random numbers in a verifiable way (figure 8.5 illustrates this concept). The following shows how this works:

1. You generate a key pair and publish the verifying key. You also publish a public seed.

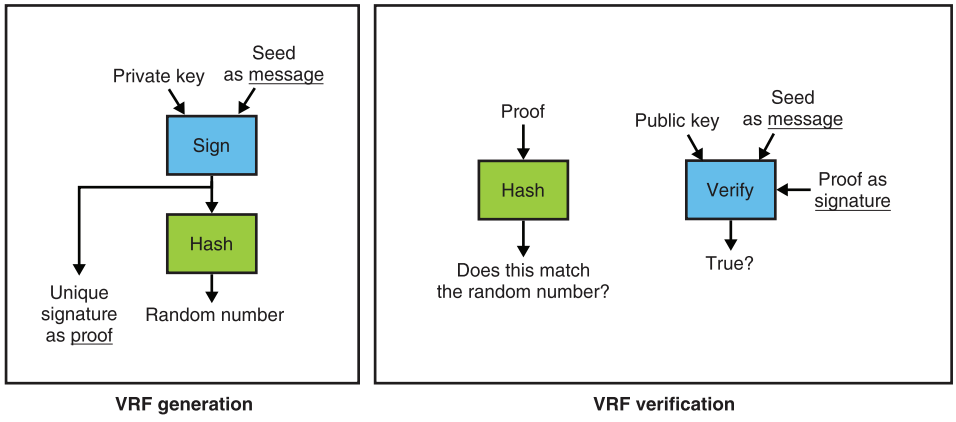



**Figure 8.5 A verifiable random function (VRF) generates verifiable randomness via public key cryptography. To generate a random number, simply use a signature scheme which produces unique signatures (like BLS) to sign a seed, then hash the signature to produce the public random number. To validate the resulting randomness, make sure that the hash of the signature is indeed the random number and verify the signature over the seed.**

2. To generate random numbers, you sign the public seed and hash the signature. The digest is your random number, and the signature is also published as proof.
3. To verify the random number, anyone can hash the signature to check if it matches the random number and verify that the signature is correct with the public seed and verifying key.

This construction can be extended to produce many random numbers by using the public seed like a counter. Because the signature is unique and the public seed is fixed, there is no way for the signer to generate a different random number.

**Exercise**

Signature schemes like BLS (mentioned in figure 8.5 and in chapter 7) produce unique signatures, but this is not true for ECDSA and EdDSA. Do you see why?

To solve this, the Internet Draft (a document that is meant to become an RFC) https://tools.ietf.org/html/draft-irtf-cfrg-vrf-08 specifies how to implement a VRF using ECDSA. In some scenarios (for example, a lottery game), several participants might want to randomly decide on a winner. We call them *decentralized randomness beacons* as their role is to produce the same verifiable randomness even if some participants decide not to take part in the protocol. A common solution is to use the previously discussed VRFs, not with a single key but with a *threshold distributed key*, a key that is split among many participants and that produces a unique valid signature for a given message only after a threshold of participants have signed the message. This might sound a bit confusing as this is the first time I've talked about distributed keys. Know that you will learn more about these later in this chapter.

One popular decentralized randomness beacon is called *drand* and is run in concert by several organizations and universities. It is available at https://leagueofentropy.com.

> *The main challenge in generating good randomness is that no party involved in the randomness generation process should be able to predict or bias the final output. A drand network is not controlled by anyone of its members. There is no single point of failure, and none of the drand server operators can bias the randomness generated by the network.*
>
> —https://drand.love ("How drand works," 2021)

Now that I've talked extensively about randomness and how programs obtain it nowadays, let's move the discussion towards the role of secrets in cryptography and how one can manage those.

## 8.6 Key derivation with HKDF

PRNGs are not the only constructions one can use to derive more secrets from one secret (in other words, to stretch a key). Deriving several secrets from one secret is actually such a frequent pattern in cryptography that this concept has its own name: *key derivation*. So let's see what this is about.

A *key derivation function* (KDF) is like a PRNG in many ways, except for a number of subtleties as noted in the following list. The differences are summarized in figure 8.6.

- *A KDF does not necessarily expect a uniformly random secret (as long as it has enough entropy).* This makes a KDF useful for deriving secrets from key exchange output, which produce high entropy but biased results (see chapter 5). The resulting secrets are, in turn, uniformly random, so you can use these in constructions that expect uniformly random keys.
- *A KDF is generally used in protocols that require participants to rederive the same keys several times.* In this sense, a KDF is expected to be deterministic, while PRNGs sometimes provide backward secrecy by frequently reseeding themselves with more entropy.
- *A KDF is usually not designed to produce a LOT of random numbers.* Instead, it is normally used to derive a limited number of keys.

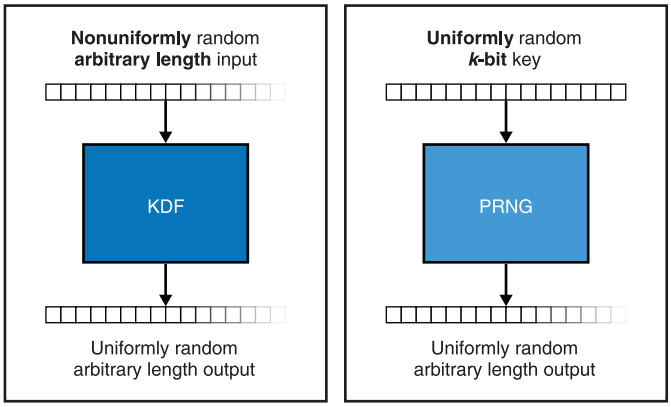



**Figure 8.6 A key derivation function (KDF) and a PRNG are two similar constructions. The main differences are that a KDF does not expect a fully uniformly random secret as input (as long as it has enough entropy) and is usually not used to generate too much output.**

The most popular KDF is the *HMAC-based key derivation function* (HKDF). You learned about HMAC (a MAC based on hash functions) in chapter 3. HKDF is a light KDF built on top of HMAC and defined in RFC 5869. For this reason, one can use HKDF with different hash functions, although, it is most commonly used with SHA-2. HKDF is specified as two different functions:

- *HKDF-Extract*—Removes biases from a secret input, producing a uniformly random secret.
- *HKDF-Expand*—Produces an arbitrary length and uniformly random output. Like PRNGs, *it expects a uniformly random secret as input* and is, thus, usually ran after HKDF-Extract.

Let's look at HKDF-Extract first, which I illustrate in figure 8.7. Technically, a hash function is enough to uniformize the randomness of an input byte string (remember, the output of a hash function is supposed to be indistinguishable from random), but HKDF goes further and accepts one additional input: a *salt*. As for password hashing, a salt

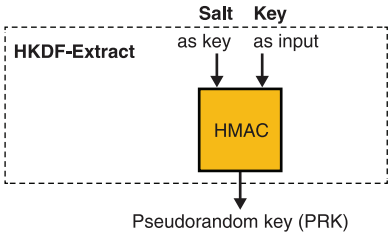



**Figure 8.7 HKDF-Expand is the second function specified by HKDF. It takes an optional `info` byte string and an input secret that needs to be uniformly random. Using different `info` byte strings with the same input secret produces different outputs. The length of the output is controlled by a `length` argument.**

differentiates different usages of HKDF-Extract in the same protocol. While this salt is optional and set to an all-zero byte string if not used, it is recommended that you do use it. Furthermore, HKDF does not expect the salt to be a secret; it can be known to everyone, including adversaries. Instead of a hash function, HKDF-Extract uses a MAC (specifically HMAC), which coincidentally has an interface that accepts two arguments.

Let's now look at HKDF-Expand, which I illustrate in figure 8.8. If your input secret is already uniformly random, you can skip HKDF-Extract and use HKDF-Expand.

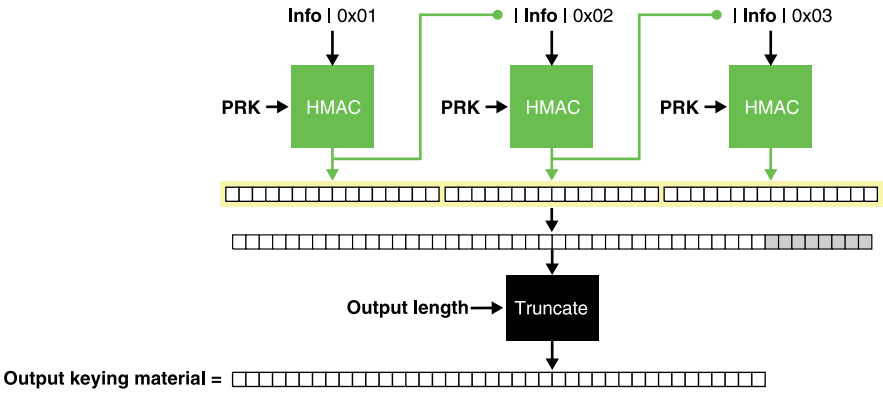



**Figure 8.8 HKDF-Extract is the first function specified by HKDF. It takes an optional salt that is used as the key in HMAC and the input secret that might be nonuniformly random. Using different salts with the same input secret produces different outputs.**

Similar to HKDF-Extract, HKDF-Expand also accepts an additional and optional customization argument called `info`. While a salt is meant to provide some domain separation between calls within the same protocol for HKDF (or HKDF-Extract), `info` is meant to be used to differentiate your version of HKDF (or HKDF-Expand) from other protocols. You can also specify how much output you want, but keep in mind that HKDF is not a PRNG and is not designed to derive a large number of secrets. HKDF is limited by the size of the hash function you use; more precisely, if you use SHA-512 (which produces outputs of 512 bits) with HKDF, you are limited to 512 × 255 bits = 16,320 bytes of output for a given key and an `info` byte string.

Calling HKDF or HKDF-Expand several times with the same arguments, except for the output length, produces the same output truncated to the different length requested (see figure 8.9). This property is called *related outputs* and can, in rare scenarios, surprise protocol designers. It is good to keep this in mind.

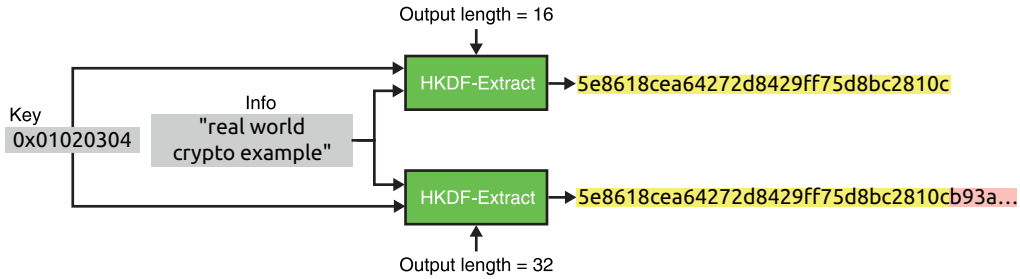



**Figure 8.9 HKDF and HKDF-Expands provide related outputs, meaning that calling the function with different output lengths truncates the same result to the requested length.**

Most cryptographic libraries combine HKDF-Extract and HKDF-Expand into a single call as figure 8.10 illustrates. As usual, make sure to read the manual (in this case, RFC 5869) before using HKDF.

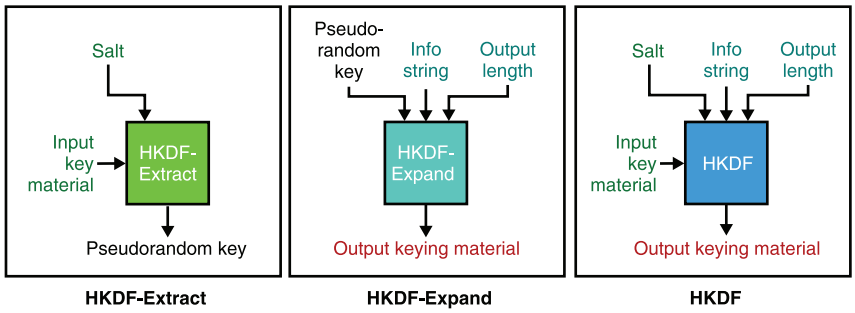



**Figure 8.10 HKDF is usually found implemented as a single function call that combines both HKDF-Extract (to extract uniform randomness from an input key) and HKDF-Expand (to generate an arbitrary length output).**

HKDF is not the only way to derive multiple secrets from one secret. A more naive approach is to use *hash functions*. As hash functions do not expect a uniformly random input and produce uniformly random outputs, they are fit for the task. Hash functions are not perfect, though, as their interface does not take into account *domain separation* (no customization string argument) and their output length is fixed. Best practice is to avoid hash functions when you can use a KDF instead. Nonetheless, some well-accepted

algorithms do use hash functions for this purpose. For example, you learned in chapter 7 about the Ed25519 signature scheme that hashes a 256-bit key with SHA-512 to produce two 256-bit keys.

> **Do these functions really produce random outputs?**
> In theory, a hash function's properties do not say anything about the output being uniformly random; the properties only dictate that a hash function should be collision resistant, pre-image resistant, and second pre-image resistant. In the real world, though, we use hash functions all over the place to implement random oracles (as you learned in chapter 2), and thus, we assume that their outputs are uniformly random. This is the same with MACs, which are, in theory, not expected to produce uniformly random outputs (unlike PRFs as seen in chapter 3), but in practice, do for the most part. This is why HMAC is used in HKDF. In the rest of this book, I will assume that popular hash functions (like SHA-2 and SHA-3) and popular MACs (like HMAC and KMAC) produce random outputs.

The extended output functions (XOFs) we saw in chapter 2 (SHAKE and cSHAKE) can be used as a KDF as well! Remember, a XOF

- Does not expect a uniformly random input
- Can produce a practically infinitely large uniformly random output

In addition, KMAC (a MAC covered in chapter 3) does not have the related output issue I mentioned earlier. Indeed, KMAC's length argument randomizes the output of the algorithm, effectively acting like an additional customization string.

Finally, there exists an edge case for inputs that have low entropy. Think about passwords, for example, that can be relatively guessable compared to a 128-bit key. The password-based key derivation functions used to hash passwords (covered in chapter 2) can also be used to derive keys as well.

## 8.7 Managing keys and secrets

All right, all good, we know how to generate cryptographic random numbers, and we know how to derive secrets in different types of situations. But we're not out of the woods yet.

Now that we're using all of these cryptographic algorithms, we end up having to maintain a lot of secret keys. How do we store these keys? And how do we prevent these extremely sensitive secrets from being compromised? And what do we do if a secret becomes compromised? This problem is commonly known as *key management*.

> *Crypto is a tool for turning a whole swathe of problems into key management problems.*
>
> —Lea Kissner (2019, http://mng.bz/eMrJ)

While many systems choose to leave keys close to the application that makes use of them, this does not necessarily mean that applications have no recourse when bad

things happen. To prepare against an eventual breach or a bug that would leak a key, most serious applications employ two defense-in-depth techniques:

- *Key rotation*—By associating an expiration date to a key (usually a public key) and by replacing your key with a new key periodically, you can "heal" from an eventual compromise. The shorter the expiration date and rotation frequency, the faster you can replace a key that might be known to an attacker.
- *Key revocation*—Key rotation is not always enough, and you might want to cancel a key as soon as you hear it has been compromised. For this reason, some systems allow you to ask if a key has been revoked before making use of it. (You will learn more about this in the next chapter on secure transport.)

Automation is often indispensable to successfully using these techniques as a well-oiled machine is much more apt to work correctly in times of crisis. Furthermore, you can also associate a particular role to a key in order to limit the consequences of a compromise. For example, you could differentiate two public keys in some fabricated application as public key 1, which is only for signing transactions, and public key 2, which is only for doing key exchanges. This allows a compromise of the private key associated with public key 2 to not impact transaction signing.

If one does not want to leave keys lying around on device storage media, hardware solutions exist that aim at preventing keys from being extracted. You will learn more about these in chapter 13 on hardware cryptography.

Finally, many ways exist for applications to delegate key management. This is often the case on mobile OSs that provide *key stores* or *key chains*, which will keep keys for you and will even perform cryptographic operations!

Applications living in the cloud can sometimes have access to cloud key management services. These services allow an application to delegate creation of secret keys and cryptographic operations and to avoid thinking about the many ways to attack those. Nonetheless, as with hardware solutions, if an application is compromised, it will still be able to do any type of request to the delegated service.

**NOTE** There are no silver bullets, and you should still consider what you can do to detect and respond to a compromise.

Key management is a hard problem that is beyond the scope of this book, so I will not dwell on this topic too much. In the next section, I go over cryptographic techniques that attempt to avoid the key management problem.

## 8.8 Decentralize trust with threshold cryptography

Key management is a vast field of study that can be quite annoying to invest in as users do not always have the resources to implement best practices, nor the tools available in the space. Fortunately, cryptography has something to offer for those who want to lessen the burden of key management. The first one I'll talk about is *secret sharing* (or *secret splitting*). Secret splitting allows you to break a secret into multiple

parts that can be shared among a set of participants. Here, a secret can be anything you want: a symmetric key, a signing private key, and so on.

Typically, a person called a *dealer* generates the secret, then splits it and shares the different parts among all participants before deleting the secret. The most famous secret splitting scheme was invented by Adi Shamir (one of the co-inventors of RSA) and is called *Shamir's Secret Sharing* (SSS). I illustrate this process in figure 8.11.

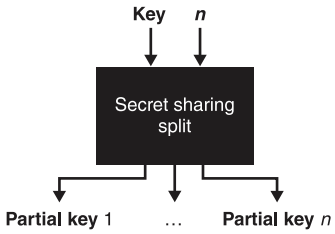



**Figure 8.11 Given a key and a number of shares *n*, the Shamir's Secret Sharing scheme creates *n* partial keys of the same size as the original key.**

When the time comes and the secret is needed to perform some cryptographic operation (encrypting, signing, and so on), all share owners need to return their private shares back to the dealer who is in charge of reconstructing the original secret. Such a scheme prevents attackers from targeting a single user as each share is useless by itself and, instead, forces attackers to compromise all the participants before they can exploit a key! I illustrate this in figure 8.12.

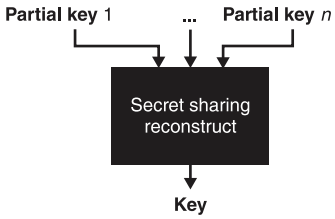



**Figure 8.12 The Shamir's Secret Sharing scheme used to split a secret in *n* partial keys requires all of the *n* partial keys to reconstruct the original key.**

The mathematics behind the scheme's algorithm are actually not too hard to understand! So let me spare a few paragraphs here to give you a simplified idea of the scheme.

Imagine a random straight line on a 2-dimensional space, and let's say that its equation—$y = ax + b$—is the secret. By having two participants hold two random points on the line, they can collaborate to recover the line equation. The scheme generalizes to polynomials of any degree and, thus, can be used to divide a secret into an arbitrary number of shares. This is illustrated in figure 8.13.

Secret splitting is a technique often adopted due to its simplicity. Yet, in order to be useful, key shares must be gathered into one place to recreate the key each and every time it is used in a cryptographic operation. This creates a window of opportunity in which the secret becomes vulnerable to robberies or accidental leaks, effectively getting

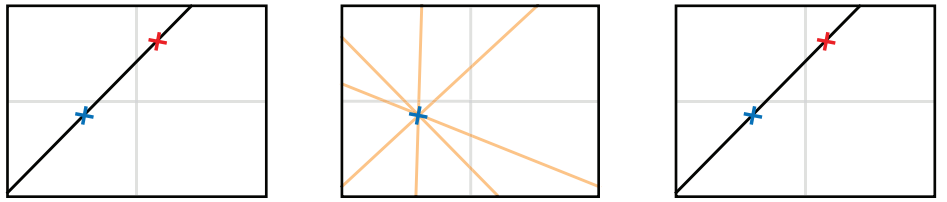



**Figure 8.13 The idea behind the Shamir's Secret Sharing scheme is to see a polynomial defining a curve as the secret and random points on the curve as partial keys. To recover a polynomial of degree $n$ that defines a curve, one needs to know $n + 1$ points on the curve. For example, $f(x) = 3x + 5$ is of degree 1, so you need any two points $(x, f(x))$ to recover the polynomial, and $f(x) = 5x^2 + 2x + 3$ is of degree 2, so you need any three points to recover the polynomial.**

us back to a *single point of failure* model. To avoid this single point of failure issue, there exist several cryptographic techniques that can be useful in different scenarios.

For example, imagine a protocol that accepts a financial transaction only if it has been signed by Alice. This places a large burden on Alice, who might be afraid of getting targeted by attackers. In order to reduce the impact of an attack on Alice, we can, instead, change the protocol to accept (on the same transaction) a number $n$ of signatures from $n$ different public keys, including Alice's. An attacker would have to compromise all $n$ signatures in order to forge a valid transaction! Such systems are called *multi-signature* (often shortened as *multi-sig*) and are widely adopted in the cryptocurrency space.

Naive multi-signature schemes, though, can add some annoying overhead. Indeed, the size of a transaction in our example grows linearly with the number of signatures required. To solve this, some signature schemes (like the BLS signature scheme) can compress several signatures down to a single one. This is called *signature aggregation*. Some multi-signature schemes go even further in the compression by allowing the $n$ public keys to be aggregated into a single public key. This technique is called *distributed key generation* (DKG) and is part of a field of cryptography called *secure multi-party computation*, which I will cover in chapter 15.

DKG lets $n$ participants collaboratively compute a public key without ever having the associated private key in the clear during the process (unlike SSS, there is no dealer). If participants want to sign a message, they can then collaboratively create a signature using each participant's private shares, which can be verified using the public key they previously created. Again, the private key never exists physically, preventing the single point of failure problem SSS has. Because you saw Schnorr signatures in chapter 7, figure 8.14 shows the intuition behind a simplified Schnorr DKG scheme.

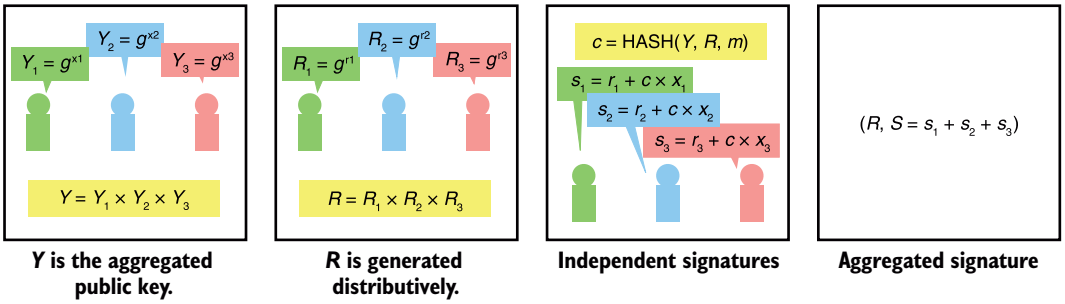



**Figure 8.14 The Schnorr signature scheme can be decentralized into a distributed key generation scheme.**

Finally, note that

- Each scheme I've mentioned can be made to work even when only a threshold $m$ out of the $n$ participants take part in the protocol. This is really important as most real-world systems must tolerate a number of malicious or inactive participants.
- These types of schemes can work with other asymmetric cryptographic algorithms. For example, using threshold encryption, a set of participants can collaborate to asymmetrically decrypt a message.

I review all these examples in figure 8.15.

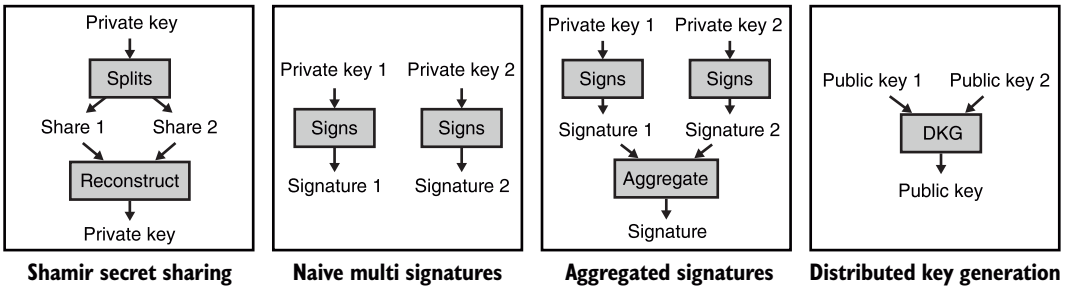



**Figure 8.15 A recap of existing techniques to split the trust we have in one participant into several participants.**

Threshold schemes are an important new paradigm in the key management world, and it is a good idea to follow their development. NIST currently has a threshold cryptography group, which organizes workshops and has the intent to standardize primitives and protocols in the long run.

## Summary

- A number is taken uniformly and at random from a set if it was picked with equal probability compared to all the other numbers from that set.
- Entropy is a metric to indicate how much randomness a byte string has. High entropy refers to byte strings that are uniformly random, while low entropy refers to byte strings that are easy to guess or predict.
- Pseudorandom number generators (PRNGs) are algorithms that take a uniformly random seed and generate (in practice) a nearly infinite amount of randomness that can be used for cryptographic purposes (as cryptographic keys, for example) if the seed is large enough.
- To obtain random numbers, one should rely on a programming language's standard library or on its well-known cryptographic libraries. If these are not available, operating systems generally provide interfaces to obtain random numbers:
  - Windows offers the `BCryptGenRandom` system call.
  - Linux and FreeBSD offer the `getrandom` system call.
  - Other Unix-like operating systems usually have a special file called /dev/urandom that exhibites randomness.
- Key derivation functions (KDFs) are useful in scenarios where one wants to derive secrets from a biased but high entropy secret.
- HKDF (HMAC-based key derivation function) is the most widely used KDF and is based on HMAC.
- Key management is the field of keeping secrets, well, secret. It mostly consists of finding where to store secrets, proactively expiring and rotating secrets, figuring out what to do when secrets are compromised, and so on.
- To lessen the burden of key management, one can split the trust from one participant into multiple participants.